4 Abstraction procédurale et récursion

4.1 Abstraction procédurale

Des sous-procédures peuvent servir à clarifier un programme. (Sous-procédures dans le sens “être appellée par un autre fonction” et pas nécessairement dans le sens d’une fonction locale.)

Exemple:

Pour tester si un état `state` est un état final d’un automate `fsa`, on peut faire

```
(member state (cadr fsa))
```

Mais cela ne montre pas très clair ce que ce teste fait. Mieux:

```
(defun final-states (fsa) (cadr fsa))

(defun is-final (state fsa) (member state (final-states fsa)))
```

Après cela on peut tester si `state` est un état final de `fsa` en utilisant

```
(is-final state fsa)
```

En plus, des telles définitions peuvent servir à renfermer une structure de données et ses procédures. Quand on décide par exemple de modifier le format d’un automate on n’a qu’à modifier les fonctions qui accèdent directement l’automate, dans ce cas c’est seulement `final-states`. Le teste `(is-final state fsa)` reste le même.

Des sous-procédures peuvent être utilisées pour extraire des parties d’une fonction qui occurrent plusieurs fois au cours de la fonction.

Exemple:

```
(defun count-un-le (l)
  (cond ((null l) 0)
        ((eq (car l) 'un) (+ 1 (count-un-le (cdr l))))
        ((eq (car l) 'le) (+ 1 (count-un-le (cdr l))))
        (T (count-un-le (cdr l)))))
```

Mieux:

```
(defun is-first-element (element l)
   (eq element (car l)))

(defun count-un-le (l)
  (cond ((null l) 0)
        ((is-first-element 'un l) (+ 1 (count-un-le (cdr l))))
        ((is-first-element 'le l) (+ 1 (count-un-le (cdr l))))
        (T (count-un-le (cdr l)))))
```

Un cas spécial d’abstraction de procédures est la récursion. Une procédure qui est appellée à l’intérieur de soi-même peut être formulée en une procédure récursive.

Exemple: fonction `count-members` qui prend une liste d’atomes `compare-list` et une liste `l` comme arguments et qui retourne le nombre d’occurrences d’éléments de `compare-list` dans `l` ou une des sous-listes de `l`.

```
> (count-members '(a b) '(a (b c) ((a))))
3
```

Définition possible de `count-members`:

```
(defun count-members (compare-list l)
   (cond ((null l) 0)
         ((atom (car l))
          (if (member (car l) compare-list)
              (+ 1 (count-members compare-list (cdr l)))
              (count-members compare-list (cdr l))))
         (T (+ (count-members compare-list (car l))
               (count-members compare-list (cdr l))))))
```

En général, il y a les raisons suivantes pour remplacer un partie d'une fonction par l'appel d'une procédure supplémentaire:

1. Clarification de la sémantique de cette partie de la fonction avec en même temps abstraction de la façon spécifique de laquelle cette partie de la fonction est réalisée. (En remplaçant par exemple `(member state (cadr fsa))` par `(is-final state fsa)`).
2. Renfermer des structures de données et des procédures qui opèrent sur ces structures de données.
3. Abstraction et extraction d'une sous-procédure de la fonction qui est appliquée plusieurs fois au cours de la fonction.
4. Formulation d'un procès recursif.

4.2 Récursion

On peut distinguer différents types de procédures récursives.

- Des "vraies" récursions comme dans `count-members` ou le systeme LISP est obligé de retourner au niveau d'un appel après avoir evalué l'appel recursif.

  Ici on distingue en plus entre fonctions avec un seul appel récursif (comme `count-un-le`) (*singly recursive*) et fonctions avec plusieurs appels récursifs (comme `count-members`) (*doubly recursive*).
- Des récursions à la fin d'un appel de fonction qui peuvent être procédées d'une façon iterative comme dans

  ```
  (defun search-det (l)
    (cond ((null l) nil)
          ((or (eq 'un (car l)) (eq 'le (car l))) T)
          (T (search-det (cdr l)))))
  ```

  Ces récursions s'appellent reécursion terminale (*tail recursion*).

Des récursions simples (single recursion) occurrent plutôt avec des structures linéaires comme des listes. Tandis que des doubles récursions occurrent plutôt avec des structures ramifiantes commes des arbres.

En général, le premier type de récursions est plus complexe.

Parfois, on peut modifier une récursion du premier type en une récursion terminale, surtout quand la récursion est singly recursive:

Version récursive:

```
(defun mylength-rec (l) (if (null l) 0 (+ 1 (mylength-rec (cdr l)))))
```

Version itérative:

```
(defun mylength-it (l) (mylength-help 0 l))

(defun mylength-help (res l)
            (if (null l) res (mylength-help (+ 1 res) (cdr l))))
```

4.3 Paramètres optionnels

La seule raison pour définir `mylength-help` étaient qu'on n'a pas voulu que la fonction principale, `mylength-it`, aie une variable `res`.

LISP permet de mettre des paramètres optionnels dans les définitions de fonctions. Syntaxe:

```
(defun <fonction> (<par non-optionnels> &optional <par optionnels>)
 ...)
```

Tous les paramètres suivants `&optional` sont optionnels.

Une fonction doit être appellée avec des valeurs pour tous ces paramètres non-optionnels. Si elle est appellée avec des valeurs de plus, ces valeurs-là sont liées aux paramètres optionnels dans un ordre de gauche à droite. Si un paramètre optionnel ne reçoit pas de valeur dans un appel, sa valeur va être `nil`.

```
(defun mylength (l &optional res)
  (cond ((not res) (mylength l 0))
        ((null l) res)
        (T (mylength (cdr l) (+ 1 res)))))
```

On peut aussi, dans une définition de fonction, donner une valeur default à un paramètre optionnel en mettant entre parenthèses le paramètre et sa valeur default.

```
(defun mylength (l &optional (res 0))
  (if (null l) res (mylength (cdr l) (+ 1 res))))
```

Au lieu de `&optional` on peut aussi utiliser d'autres paramètres. Un paramètre utile est `&rest` qui permet de définir des fonctions avec un nombre arbitraire d'arguments. Si le dernier paramètre dans la définition d'une fonction est prècedé par `&rest`, la fonction accèpte pour ce dernier paramètre un nombre arbitraire d'arguments qui sont mis dans une liste liée à ce dernier paramètre.

```
> (defun nombre-arguments (&rest args) (length args))
NOMBRE-ARGUMENTS
> (nombre-arguments 'a 'b 'c 'd 'e)
5
> (nombre-arguments)
0
```

4.4 Lambda abstraction

`defun` définit une fonction en lui donnant en même temps un nom. La fonction `lambda` sert à définir une fonction sans lui donner un nom, une fonction anonyme.

Syntaxe:

```
(lambda (<parametres>)
  <corps de la fonction>)
```

lambda cause une abstraction fonctionnelle du <corps de la fonction> avec <parametres> comme paramètres.

Par exemple:

(lambda (l) (caddr l)) engend une fonction qui prend une liste comme argument et qui en retourne le troisième élément.

```
> ((lambda (l) (caddr l)) '(a b c d))
C
```

Un appel d'une fonction définie avec lambda constitue des liaisons temporaires et est équivalent à une expression avec let:

```
((lambda (<par 1> ... <par n>)
         <corps>)
 <arg 1> ... <arg n>)
```

est équivalent à:

```
(let ((<par 1> <arg 1>) ... (<par n> <arg n>))
     <corps>)
```

lambda est en fait la forme originale, let est définie à l'aide de lambda et facilite l'expression de liaisons temporaire.

lambda est utile qund on veut passer une fonction comme argument à une fonction.

Exemple: la fonction mapcar prend une fonction et une liste et elle applique la fonction à tous les éléments de la liste. Elle retourne la liste des résultats.

```
> (mapcar #'car '((a b) (c d)))
(A C)
> (mapcar #'(lambda (x) (+ 1 x)) '(1 2 3))
(2 3 4)
```

Pour obtenir une procédure sans l'appeller toute suite, on utilise #' devant le nom de la procédure ou devant l'expression lambda qui définit la procédure.

D'autres fonctions qui prennent des fonctions comme arguments sont funcall et apply.

funcall prend comme arguments une fonction et des s-expressions et applique son premier argument à ses autres arguments.

```
(funcall #'<fonction> <arg1> ... <arg n>)
```

est équivalent à

```
(<fonction> <arg 1> ... <arg n>)
```

Exemple:

```
> (car '(a b c))
A
> (funcall #'car '(a b c))
A
```

On utilise `funcall` quand on a lié une fonction à une variable et on veut appeller cette fonction en utilisant la variable. Par exemple dans l'appel d'une fonction avec une fonction comme argument, cette dernière fonction est liée à un des paramètres de la fonction principale. Quand, à l'intérieur de cette fonction principale, on veut appeller la fonction-argument, on utilise `funcall`:

```
> (defun application (fonction liste) (funcall fonction liste))
APPLICATION
> (application #'car '(a b c))
A
```

Deuxième exemple, avec une récursion terminale: une fonction `extract` qui extrait tous les éléments d'une liste pour lesquels un prédicat `pred` est vrai.

```
> (defun extract (pred liste &optional (resultat ()))
   (cond ((null liste) (reverse resultat))
         ((funcall pred (car liste))
          (extract pred (cdr liste) (cons (car liste) resultat)))
         (T (extract pred (cdr liste) resultat))))
EXTRACT
> (extract #'(lambda (x) (or (eq 'a x)(eq 'b x))) '(a a b c a d))
(A A B A)
```

`apply` fait plus ou moins la même chose que `funcall` sauf qu'il prend comme arguments la fonction et la liste de ses arguments (au lieu de ses arguments comme dans `funcall`).

```
(apply #'<fonction> (list <arg1> ... <arg n>))
```

est équivalent à

```
(funcall #'<fonction> <arg1> ... <arg n>)
```

Exemple:

```
> (apply #'car '((a b c)))
A
```